What Is Aspect-Oriented Programming, Revisited

Robert E. Filman
Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA 94035
rfilman@arc.nasa.gov

Abstract

For the Advanced Separation of Concerns workshop at OOPSLA 2000 in Minneapolis, Dan Friedman and I wrote a paper [10] that argued that the distinguishing characteristic of Aspect-Oriented Programming systems (qua programming systems) is that they provide quantification and obliviousness. In this paper, I expand on the themes of our Minneapolis workshop paper, respond to some of the comments we've received on that paper, and provide a computational formalization of the notion of quantification.

1. Quantification and Obliviousness

For the Advanced Separation of Concerns workshop at OOPSLA 2000 in Minneapolis, Dan Friedman and I wrote a paper [10] that argued that the distinguishing characteristic of Aspect-Oriented Programming systems (qua programming systems) is that they provide quantification and obliviousness. *Quantification* is the idea that one can write unitary and separate statements that have effect in many, non-local places in a programming system; *obliviousness*, that the places these quantifications applied did not have to be specifically prepared to receive these enhancements.

Obliviousness states that you can't tell that the aspect code will execute by examining the body of the base code. Obliviousness is desirable because it allows greater separation of concerns in the system creation process—concerns can be separated not only in the structure of the system, but also in the heads of the creators. But is it essential for AOP? A non-oblivious system requires a program that addresses some concern to include some notation or action that invokes the code for that concern. Conventional programming languages provide straightforward mechanisms—primarily subprograms, but also inheritance—for separating concerns. If one argues that AOP is needed because subprogram calls are too expensive, I would suggest that this is an occasion for improving conventional compilation technology, not the invention of new programming language constructs. If one argues that AOP is needed because "conventional languages," (which these days seems to mean Java) lack the ability to express first class functions, exception routines and closures, well, that's a problem with Java, not a need to introduce new language structures, or perhaps an argument that AOP is just a response to the limitations of Java. To the extent that one argues that AOP is needed because the caller will not always remember to call the designated routine, well, that's an argument for obliviousness. (This is not to ignore the downside of obliviousness, that systems melded from separate minds may not function the way anyone intended and that systems composed by formal rules may produce surprising behavior. Nor is it the assertion that AOP techniques must always be used obliviously—there's no great harm in knowing what's going on, either. The argument is that AOP provides obliviousness, which is interesting, and without it, it's not that special.)

Let us turn to quantification. AOP is a linguistic mechanism for expression cross-cutting concerns. One can't help noticing that there's a certain circularity to this definition: A concern is cross-cutting only if the existing linguistic mechanisms (and the chosen program structuring)

make it difficult to express the concern monolithically. So AOP is a linguistic mechanism for dealing with weak languages—in particular, the tendency of imperative and functional languages to express programs as a sequence of actions. Crosscutting concerns don't phrase themselves as neat, monolithic sequences. The original paper gave examples of inherently quantified languages such as rule-based and logic systems. Such languages revel in "crosscutting concerns." For example, a selling point of early knowledge-based systems was the feature of being able to just "throw in" additional knowledge and having it take effect. Separation of concerns wasn't the problem—it was congealing concerns. The issues in these systems centered on sequencing concerns—what got to run first, and which things would be excluded altogether.

What is quantification? Informally, we have the idea that some programmer has written some code that results in a single thread performing actions *x*, *y*, and *z*. Since this is conventional code, the programmer has used some conventional constructs to express this intent—for example, *x*, *y*, and *z* may be sequential statements in the program code; *x* and *z* may be the loop test and increment of a loop containing *y*; *x*, *y*, and *z* may be components of an expression evaluation; or *y* may be a procedure call inserted between *x* and *z*. In each of the cases but the last, *x*, *y*, and *z* are textually localized; in the last, the programmer has knowingly (unobliviously) invoked distant action *y*. (Object-orientation has loosened his knowledge of what exactly *y* is.) Conventional forms of obliviousness slip in when, for example, one of the actions is an automatic type conversion or the implicit invocation of the super constructor, as inserted by the language compiler.

Actions *x*, *y*, and *z* came to be executed because the programmer wrote method *m* in class definition *c* (or the equivalent) and handed the program to the compiler and class-loader (among other actions). (For simplicity, read *x*, *y*, and *z* as a shorthand for the entirety of *m*.) When it came time for the program to execute, the interpreter (and something was interpreting something in this stored-program machine) noticed the invocation of *m* and started performing actions *x*, *y*, and *z*. This is not an assertion about what mechanism the AOP system used to accomplish this result—it could have arisen from a compiler [12, 13], by wrapping [1, 5, 9], by load-time transformation [4], by a metainterpreter [6], or by any number of mechanisms that haven't been invented yet. It's an argument that separately specified behaviors have come to be executed together, and that some of these separate specifications were about groups of "join points."

Can I formalize this notion? Perhaps informally. If one thinks about this in terms of implementing the system entirely by an interpreter[1], the action of defining *m* has stored in an environment of the interpreter the association between *m* and *x*, *y*, and *z*. In the execution of the interpreter, retrieving *m* has returned *x*, *y*, and *z*. This interpreter is exhibiting *preservation of assignment*: if you store something in a location, and nothing comes along to change that assignment, then when you retrieve the value, you get back what you put in.[2]

[1] A good way to understand the semantics of a programming language is by careful examination of an interpreter that implements that language [11]. One typically feeds such an interpreter an expression that consists of a set of definitions of symbols (classes, methods, functions and variables) together with an expression to be evaluated (e.g., the body of a "main"). The interpreter proceeds to determine the value of that main expression, given the definitions. Classical interpreters do things like build environments to store values and consult these environments in determining the value of expressions and subexpressions. For example, our interpreter is likely to build an environment where each of the outer definition's symbols is bound to a structure representing the body of the definition, and to build derivative environments (or modify existing environments) when faced with a subprogram call or local variable declaration. Differences in language semantics are typically centered on how and when environments are built, shared and modified, and to which elements of the process (e.g., the interpreter itself) the program has access. So declaring "int x" typically reserves space in an environment for x; assigning "x ←3" puts "3" in that space, and using "x" in "x + 2" retrieves the "3" for "x".

[2] Note the correspondence to Floyd-Hoare semantics: After "x ← 3", we can assert "x = 3". Note also that we have glibly used the same notion of assignment for the base variables in the language as for the subpro-

Imagine a join point at *y*, where aspect *a* is to execute before *y*. To make this happen, the programmer has likely made an assertion of the following form: "In situations that match some condition (of which *y* is one), 'before' aspect *a* applies. In this situation, when our interpreter goes for the value of *m*, it gets back *x*, *a*, *y*, *z*. But there has not been an assertion of that value—that is, no user statement said, "set *m* to something that evaluates to *x*, *a*, *y*, *z*" Rather, the system has combined the base assertion: "*m* is *x*, *y*, and *z*" with the quantified statement about a pointcut *P*: "whenever (some) *P* is true, do *a* first" and the fact *P* (*y*), to produce the *x*, *a*, *y*, *z* value. We thus understand that the interpreter of an Aspect-Oriented Programming system is characterized by not preserving assignment. Rather, such an interpreter has done some theorem proving—it has instantiated a universal quantified formula about when an aspect applies with a particular piece of code, and implicitly redefined a method definition with the result. A key element of the peculiarity of this (in contrast to preservation of assignment) is that the base assertion and the aspect assertion can occur in either order, and that the aspect assertion can modify many otherwise unrelated base assertions at once.[3] Note that this is not a claim that the interpreter needs to be reflective or engage in meta-programming, any more than an ordinary compiler needs to be reflective or engage in meta-programming—the theorem proving can all take place at compile/loading time, and not during the actual run of the program. The theorem proving is also likely to be over the "closed world" of the specific program in question, and not all programs, and thus quite computationally tractable.

We can't put any other constraints on the process—that is, we can't place other restrictions on the behavior of an AOP interpreter.[4] If you put a handkerchief in a hat, wait a while, and pull out a rabbit, we don't know if the act of putting the handkerchief in the hat turned it into a rabbit, if it turned into a rabbit through the act of pulling it out, or if something wandered around at some point in between and effected the metamorphosis. We also don't know if there's a handkerchief inside the rabbit, or if the molecules of the handkerchief have been rearranged to form something that's newly rabbit. Correspondingly, (to rephrase the argument in familiar terms), we don't know if some compiler has integrated the aspect and base code before generating the class file, if a transformation happened to the byte-code in the class file, or if the underlying virtual machine noticed just as it was about to present the handkerchief that it really ought to show a rabbit. These are all mechanisms that have been used (metaphorically speaking, of course) to build AOP systems.

2. Waterfalls

AOP is often explained in terms of crosscutting concerns. But to get to the point of having crosscutting concerns, one needs to have something to crosscut. Concerns are a primitive in the system development process—they're something one cares about. There's much overlap between "concerns" and "requirements," except we probably won't use the term "concern" for a requirement

---

gram definitions. It turns out that this makes sense; building the interpreter crystallizes how much sense it makes. Making this choice does not bind us to using the same namespace, shadowing, or contextual rules for variables and subprograms. It merely notes that subprograms have bindings (bodies and parameters) and that invoking a subprogram implies examining that binding.

[3] Certain varieties of inheritance share this property such as, "before" and "after" methods in mixins. This echoes our argument in the original paper that certain kinds of inheritance could be used as early Aspect-Oriented Programming languages. However, I note a mistake in the original paper, where mixins in multiple inheritance systems were cited as an early form of AOP, given the ability to easily modify an inheritance hierarchy external to the actual objects. I had my "AI blinders" on, thinking of things like KEE [7] (my most liberal use of multiple inheritance and mixins), and forgot that in the conventional universe an object's class is built-in, not added later.

[4] And I suspect somebody will be able to come up with a pathological interpreter for AOP that doesn't act this way at all.

that's too specific (e.g., "Sales tax rate is to be determined by looking up the purchaser's home county in the following table") and are perfectly happy to introduce concerns anywhere in the development process (e.g., a concern about an implementation detail, such as prefetching [1], as opposed to the classic waterfall, which assumes that the requirements have magically appeared at the beginning of the process, and relate to the external world, as opposed to the emerging system structure.) We don't really know if concerns crosscut until we've gone far enough down the river to have a design.

In general, to meet some set of concerns *C*, one can create, limited by the mechanisms of one's programming environment, solution *S*. *S* will have some structure (for example, classes, modules and methods.) We can identify four different kinds of concerns [8]:

- *Local* concerns that talk about a specific spot in *S*. For example, authenticating a user might happen just in the login routine.
- *Systematic* concerns apply in several or many places in *S*. They require "doing the right thing" at each such spot—typically, executing the code that realizes the concern. For example, checking the access control rights of an (authenticated) user to particular data or functionality might be required in many modules.
- *Combinatoric* concerns that are computationally intractable expressions of overall system behavior. For example, establishing that users cannot use the existing code to access functionality unless they have the appropriate authentication might require proving properties of the system as a whole, independent of the properties of particular modules.
- *Aesthetic* concerns that express non-computable qualities of the system. For example, establishing that there is no covert channel by which information can leak from the system to unauthorized users may not be computable.

Conventional mechanisms do a good job of supporting the first of these—in the place in the code where you've got the concern, write the code to handle the concern. The last two are difficult to automate in any case.

AOP is useful for handling the second kind of concern, ones where the appropriate hygiene (systematic behavior) can achieve the goal. AOP systems naturally quantify over the systematic loci and apply the appropriate aspects—assuming, of course, that the systematic loci are join points of the AOP system, and the quantification language is rich enough to describe them as a set.

I also note that AOP is about programming *languages* and environments: a crosscutting concern in one environment can be part of the scenery in another. For example, using Enterprise Java Beans, security and transactions are built-in in the environment (and thus, if you accept the built-in solutions) not cross-cutting concerns.

3. Problem Spaces

Understanding something involves both understanding how it works (*mechanism*) and what it's good for (*methodology*). In Computer Science, we're rarely shy about grandiose methodological claims (see, for example, the literature of AI or the Internet). But mechanism is important — appreciating mechanisms leads to improved mechanisms, recognition of commonalities and isomorphisms, and plain old clarity about what's actually happening. The understanding gained by examination of mechanism leads to generalization and richer systems. The quantification and obliviousness argument is about looking for the common mechanism in AOP systems, seeing how to understand particular AOP systems with respect to those mechanisms, and looking for the places where those mechanisms can be generalized. We're trying to answer the question, "Is *X* an AOP system?" where the answer is not based on whether the creator of the system called it AOP,

but rather on whether its mechanisms can be used to do AOP.[5] Here we take inspiration from Wegner [14], who early in the OO discussion tried to clarify the dimensions of OO language design. We need to be able to exclude non-AOP languages from the definition of AOP, and to get a vocabulary for talking about the differences between different AOP languages.

In the original paper, we argued [10]:

> "AOP is thus the desire to make programming statements of the form
>
> *In programs* P, *whenever condition* C *arises, perform action* A.
>
> over 'conventionally' coded programs *P*."

This implies that the semantic definition of an AOP language ought to be characterized in terms of three kinds of conditions:

P: The points in the program on which actions can take place. This includes the join points, and what contextual information about the program is available to the aspects.

A: The aspects.

C: The conditions that tie the aspects to the program.

The quantification claim is that "aspect languages" have quantification conditions, C, that talk about sets of P and allow A can be defined separately from P; the obliviousness claim is that real aspect languages do not require P to mention A.

Let us compare this list with the list that came out of the Minneapolis workshop, on dimensions of AOP systems. Of the workshop concerns:

- *Obliviousness* (must the writer of the source be aware that aspects will be applied to it), and *intimacy* (does a programmer have to do anything to prepare the code for aspects) are instances of issue of obliviousness, though the last also speaks to P, the places in the program where aspects interact.
- *Join points*, and *source code encapsulation* (does the source describe the points to be composed) are descriptions of P.
- *Aspect parameterization* (the extent to which aspects can be customized for a particular use), *dominant decomposition*, *visibility* (what part of other concerns are visible to an aspect), and *symmetry* (do aspects and the main program have different privileges) are all about the kinds of actions one can perform. The last three are especially about

[5] Imagine, if you will, Anne, a Computer Scientist working in isolation, who has invented a language exactly like AspectJ, except that she believes the purpose of the language is to insert debugging statements into conventional Java code. She's called the language "DebugJ," and everywhere AspectJ uses the word "aspect" she uses "debug." Her methodological explanations are all about temporarily inserting only debugging statements into code under development. Is DebugJ an AOP language?

Consider Betty, who believes she has invented a new aspect-oriented language called "AspectTran." AspectTran looks just like Fortran, except that "subroutine" has been replaced by the word "aspect." Betty explains that users should encode concerns separately in these aspect routines, and if a concern crosscuts another, they should invoke the aspect code like a subprogram. Shared state between the aspect and the base routine is passed in parameters to the aspect call. Since Fortran is call-by-reference, one can even have an aspect change the value of a local variable! Betty is proud of how little she had to modify the Fortran compiler to create AspectTran. Is AspectTran an AOP language?

If this seems far-fetched, keep in mind how long it took to convince Ada aficionados that Ada83 was not an object-oriented language. Dahl, Myhrhaug and Nygaard never mentioned that Simula 67 was an object-oriented language. Nevertheless, it's universally acclaimed as the first. Decide for yourself if Ada95 is an object-oriented language: one can do all the object-oriented things by using particular features in specific and non-obvious ways, but the term "object" is not part of the language's lexicon.

whether aspects can play the role of P. *Monotonicity* (can aspects delete behavior) can be (semantically) understood as asserting a null or error-raising action in place of P.

- *Explicit composition language* (does the system have a separate language for describing which aspects are applied where) *object-oriented programming* (do the concepts apply to more than OO programs), *globality vs. locality* (are aspects about the program as a whole, or specific parts) are issues of expressing the quantification condition C. My workshop notes omit a classification of whether the language is providing aspects about the static description of the program and/or its dynamic behavior (the Jumping Aspects problem [2]); these would also be instances of C.

The decomposition does not do a good job of expressing conflicts (mechanisms for recognizing aspect conflicts, and, more importantly, the issue of ordering actions), and environment (the degree to which the composition and execution depend on the external state.)

The workshop items on implementation mechanisms: static vs. dynamic composition, modular compilation, deployment-in-place on an existing system, target representation (e.g., source, byte code) and verification; and software process: reusability, domain-specificity, analyzability and testability are, of course, about implementation and not semantics.

4. Concluding Remarks

Workshops always call for "position papers," and almost inevitably receive what would usually pass for scientific papers. It is pleasing to be able to have had the opportunity for so many pages to argue positions without much requirement for experimentation, system building or proofs to back them up. My thanks to the readers for their tolerance.

Acknowledgements

My thanks to Daniel Friedman, Gregor Kiczales, Diana Lee and Tarang Patel for comments on the drafts of this paper.

References

1. Aksit, M., Wakita, K., Bosch, J., Bergmans, L., and Yonezawa, A. Abstracting Object Interactions Using Composition Filters. In Guerraoui, R., Nierstraz, O. and Riveill, M, (Eds.) *Object-Based Distributed Programming, Lecture Notes in Computer Science 791*, Berlin: Springer Verlag, 1993.
2. Brichau, J., De Meuter, W., and De Volder, K. Jumping aspects. Workshop on Aspects and Dimensions of Concerns, ECOOP 2000, Cannes, France, June 2000.
3. Coady, Y., Kiczales, G., Feeley, M., Hutchinson, N., and Ong., J. S. Structuring System Aspects. *Comm. ACM, 44* (10), 2001, in press.
4. Cohen, G. Recombing Concerns: Experience with Transformation. In First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems (OOPSLA '99). http://www.cs.ubc.ca/~murphy/multid-workshop-oopsla99/position-papers/ws23-cohen.pdf
5. Constantinides, C. A., and Elrad, T. Composing Concerns with a Framework Approach. International Workshop on Distributed Dynamic Multiservice Architectures, 21st International Conference on Distributed Computing Systems (ICDCS-21). Phoenix, Arizona, April, 2001, Vol. 2, pp. 133–140.
6. Czarnecki, K., and Eisenecker, U.W. *Generative Programming: Methods, Tools, and Applications.* Boston: Addison Wesley, 2000.

7. Fikes, R. E., and Kehler, T. The role of frame-based representation in reasoning, *Comm. ACM, 28* (1985) pp. 904–920.
8. Filman, R. E., Achieving Ilities, *Workshop on Compositional Software Architectures*, Monterey, California, Jan. 1998. http://www.objs.com/workshops/ws9801/papers/paper046.doc
9. Filman, R. E., Barrett, S., Lee, D.D., and Linden, T. Inserting Ilities by Controlling Communications. *Comm. ACM,* in press. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/oif-cacm-final.pdf
10. Filman, R. E., and Friedman, D. P. Aspect-Oriented Programming is Quantification and Obliviousness. Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000, Minneapolis. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/aop-is.pdf
11. Friedman, D. P., Wand, M., and Haynes, C. T. *Essentials of Programming Languages (2nd Edition).*Cambridge, MA: MIT Press, 2001.
12. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An Overview of AspectJ, *Proceedings ECOOP 2001,* in press.
13. Ossher, H. and Tarr, P. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. *Proc. Symp. on Software Architectures and Component Technology: The State of the Art in Software Development.* Kluwer, in press.
14. Wegner, P. Dimensions of Object-Based Language Design**,** *Proceedings OOPSLA '87, ACM SIGPLAN Notices, 22* (12) December 1987, pp. 168–182.